Programozási technológia I.

Sike Sándor

Tudnivalók

- Előfeltétel: Programozás
- Számonkérés:
  - jegy (l. gyakorlatok)
- Irodalom:
  - Sike Sándor, Varga László: Szoftvertechnológia és UML
  - Ian Sommerville: Szoftverrendszerek fejlesztése

A szoftvertechnológia kialakulása

- Katonai számítások (ENIAC, JONIAC).
- Sok számítással járó tudományos és műszaki feladatok.
- Mechanikusan ismétlődő ügyviteli feladatok is, és fizikai folyamatok adatainak gyűjtése, és az eredmények kiértékelése alapján a folyamatok vezérlése (hadiipar, űrkutatás).
- Gyártósorok vezérlése, a termelés folyamatához kapcsolódó ügyviteli feladatok, a megrendelések, a megrendelésekhez szükséges raktárkészletek, a szállítások ütemezése stb.; komplex termelésirányítási rendszerek létrehozásának igénye.

A szoftvertechnológia kialakulása (folyt.)

- Szolgáltató rendszerek (banki szolgáltatások, biztosítás); illetve a PC-k megjelenésével előtérbe kerültek a kisvállalkozások, az oktatás, a játék, a szórakozás stb. igényei.
- Globális kommunikációs alkalmazások, multimédia, távmunka.
- Osztott rendszerek, hálózati alkalmazások, multimédia, mobil eszközök.

A hardver követni tudta az igényeket:

- egységek integrációja
- erőforrások kapacitásának növelése nagyságrendekkel, méret csökkentése
- illesztő egységek az ember–gép kapcsolathoz

Szoftver:

- Az egyszemélyes feladatokkal a magasan képzett szakemberek képesek voltak megbirkózni.
- A nagy rendszerek esetén, amikor több ember együttműködésével lehet befejezni fejlesztést, a létező módszerek alkalmatlanok a feladatok megoldására:
    - elkészítési időpont nem kézben tartható,
    - elkészült programokban rejtett hibák maradtak,
    - programok előállítási költsége előre megbecsülhetetlen módon növekedett.

Szoftverkrízis

A probléma megoldásához szükség volt annak a felismerésére, hogy:

- a *program termékké* vált, és mint minden termék esetében
- az *előállításához technológiára* van szükség.

Mit jelent az, hogy a program termék? Azt, hogy:

- van *szolgáltatási funkciója*,
- van *minősége*,
- van *előállítási költsége*,
- van *előállítási határideje*.

Szoftvertechnológia célja

- Funkció, minőség, költség, határidő: tervezési paraméterek.
- A szoftvertechnológiának biztosítania kell a tervezési paramétereknek megfelelő termék előállítását.
- A *szoftvertechnológia tárgya* tehát a *nagy méretű programrendszerek* előállítása.

Nagy méretű programrendszerek jellemzői

- *Nagy bonyolultságú rendszer*, azaz fejben tartva nem kezelhetők a kidolgozás során felhasználandó részletek: az objektumok, azok jellemzői, összefüggései stb.
- *Csapatmunkában* készül.
- *Hosszú élettartamú*, amelynek során számos változatát kell előállítani, azokat követni, karbantartani stb. kell.

A szoftvertechnológia célkitűzése

- Előírt minőségű programtermék,
- előre megállapított határidőre,
- előre meghatározott költségen történő előállítása.

A szoftvertechnológia összetevői

1. Módszerek a programkészítés különböző fázisai számára.
2. Szabványok (kidolgozási, dokumentációs stb.), amelyeket a program kidolgozása során kötelező betartani; és ajánlások, amelyek hozzájárulhatnak a program minőségének javításához.
3. Programeszközök, egységes rendszert alkotó programfejlesztési környezet, amelyek megkönnyítik és biztonságosabbá teszik az emberi munkát.
4. Irányítási módszerek a programkészítés folyamatának vezérlésére, szervezésére.

Az objektumelvű programozás kialakulása

Programozás fejlődése:

- gépi kód *konkrét*
- mnemonikok (assembly)
- vezérlési szerkezetek, eljárások (FORTRAN)
- adattípusok (Pascal, C)
- típusosztályok (Ada)
- Objektumosztályok (C++, Java) *absztrakt*

- Procedurális tervezés: Funkciókból indul ki, azokat dekomponálva alakul ki a szerkezet.
- Közös erőforrást (adatszerkezetet) használó műveletek csoportosítása.
- A modulokhoz szabványos hozzáférési felület tartozik. (Típus)
- Típusöröklődés.

Objektumelvű tervezés

- Nem a funkciókból, tevékenységekből indulunk ki, hanem az adatokból, a feladatban részt vevő elemekből.
- Ezeket azonosítjuk, csoportosítjuk, felderítjük kapcsolataikat, felelősségeiket. Így jönnek létre objektumok, illetve osztályok.
- A rendszer funkcionalitását az egymással együttműködő objektumok összessége adja ki. Egy objektum csak egy jól meghatározott részért felelős.
- Az objektumok adatot tárolnak, ezek kezeléséért felelősek, de ezeket elrejthetik a külvilág elől. Szabványos módon lehet az objektumokkal kapcsolatba lépni.

Eszközök

- Objektumelvű nyelvek (C++, Java, C#).
- Szabványos tervezési nyelv (UML).
- Ezekre épülő integrált fejlesztő eszközök.

Példa

LEGO építmények gyári kezelése

- Elemekből rakhatunk össze építményeket.
- Egy építmény tartalmazhat elemeket és részépítményeket vegyesen.
- Részépítmények tetszőleges mértékben egymásba ágyazhatóak.
- Feladat: egy építmény árának meghatározása, ami a benne szereplő elemek árainak összege.
- Később:
  - egy építmény elemeinek listázása,
  - egyes elemekből mennyi felhasználható elem van készleten.

Elemek

A példa szempontjából a következőket kell ismernünk egy elemről:

- név (string),
- szín,
- ár (egész).

Építmények

- Az elemek összeszerelhetők bonyolultabb szerkezetekbe, amelyeket építményeknek nevezünk.
- Egy építmény tetszőleges számú elemből állhat, és hierarchikus szerkezete lehet, vagyis tartalmazhat építményeket is.
- A tartalmazott építményt a továbbiakban részépítménynek nevezzük.
- Egy részépítményben lehetnek elemek, illetve további részépítmények.

Objektumok

- Első lépésben a rendszer objektumait fogjuk azonosítani.
- A tervezés során az egyik legnehezebb feladat a rendszer adatainak felosztása objektumok halmazába úgy, hogy az objektumok sikeresen működjenek együtt a rendszer teljes működésének megvalósításában.
- Egy gyakori ökölszabály az objektumok kiválasztására, hogy a valóságos elemeknek a modellben objektum feleljen meg.
- Rendszerünk egyik fő feladata, hogy nyomon kövesse az összes elemet.
- Ezért adódik, hogy minden elemet objektumként kezeljünk a rendszerben.

- Sokféle elemobjektum fordulhat elő, amelyek különböző elemeket írnak le, de mindegyiknek ugyanaz lesz a szerkezete.
- Ugyanazt a valóságelemet kifejező objektumhalmaz közös szerkezetét osztállyal írjuk le.
- Az objektumhalmaz minden eleme egy példánya lesz az osztálynak.
- Az osztály egyrészt tartalmazza a közös szerkezetet (adatok), másrészt az objektumokon végrehajtható műveleteket.
- Esetünkben az `Elem` osztály létrehozása a tervezés első lépése.

Elem osztály (Java)

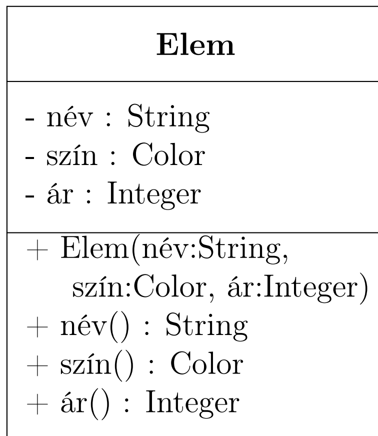
```
import java.awt.Color;

public class Elem
{
    private String  név;
    private Color   szín;
    private int     ár;

    public Elem(String név, Color szín, int ár)
    {
        this.név = név; this.szín = szín;   this.ár = ár;
    }

    public String név()     { return név; }
    public Color szín()     { return szín; }
    public int ár()         { return ár; }
}
```

Elem osztály (UML)



Az osztályok fordítási időben lesznek meghatározva, az objektumok viszont futási időben jönnek létre, mint az osztályok példányai.

```
Elem e = new Elem("2x2 kocka", Color.RED, 2);
```

művelet végrehajtása után egy új objektum jön létre.

A memóriában egy terület tartozik az objektumhoz, amely a megfelelő értékekkel rendelkezik.

e : **Elem**
név = "2x2 kocka" szín = Color.RED ár = 2

Azonosíthatóság

- Lényeges eleme az objektum definíciójának, hogy az objektumok megkülönböztethetők egymástól, azaz bármely objektum megkülönböztethető bármely más objektumtól.
- Ez akkor is teljesül, ha két objektum pontosan ugyanazokat az adatokat tartalmazza és felületük is megegyezik.

Például a következő programrészlet eredménye két objektum, amelyek állapota megegyezik, az objektumok mégis megkülönböztethetők.

```
Elem e1 = new Elem("2x2 kocka", Color.RED, 2);
Elem e2 = new Elem("2x2 kocka", Color.RED, 2);
```

Azonosíthatóság (folyt.)

- Az objektumelvű modell feltételezi, hogy minden objektumhoz tartozik egy „azonosság”, amely egyfajta címkeként megkülönbözteti az objektumot másoktól.
- Ez az azonosság egy belső, lényeges része az objektumelvű modellnek, és különbözik az objektumban tárolt adatok mindegyikétől.
- Objektumelvű nyelvek esetén az objektum memóriabeli címe használható erre a célra. Ez nyilvánvalóan eltérő különböző objektumok esetén.
- Java: objektumok egyenlőség vizsgálata azok memóriacímét veti össze. (Kivétel: `String`.)
- `this` az objektum saját memóriacíme.

Azonosíthatóság (folyt.)

- UML: az objektumokhoz neveket rendelhetünk az osztálynév mellett, és így biztosíthatjuk az objektum egyediségét.
- Ezeket a neveket a modellen belül használhatjuk, és lehetőséget adnak, hogy egy objektumra egyedileg hivatkozzunk a modellben.
- Az objektumnév nem felel meg semmilyen adategységnek.
- Az objektumnév különbözhet annak a változónak a nevétől, amellyel a programban hivatkozunk az objektumra.
- Gyakran kényelmes és praktikus, ha a két név megegyezik.
- (Ugyanakkor több, mint egy változó hivatkozhat ugyanarra az objektumra, és egy változó az élettartama során több objektumra is hivatkozhat. Ezért ez a névegyezés nem mindig valósítható meg.)

Adatismétlés

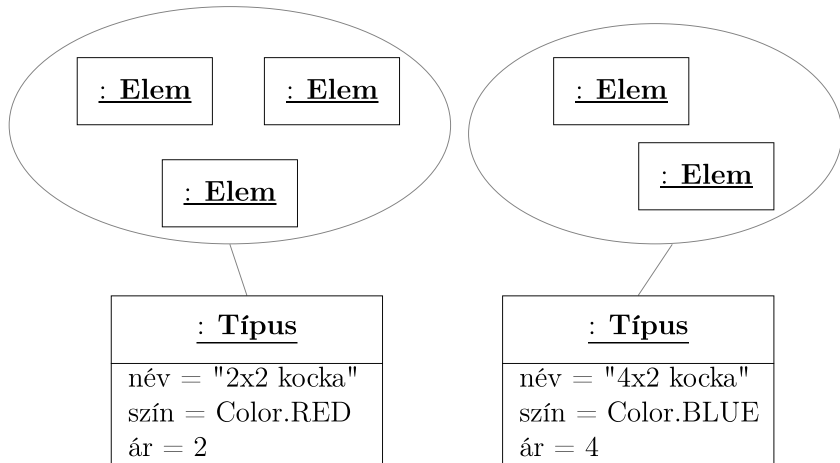
- A bevezetett modell ugyan kézenfekvő, azonban egy elemtípust leíró adatokat megismételjük minden egyes elem esetén.
- Az ismétlés oka: a leírásokat az objektumokban tároljuk.
- Ha a rendszerben kettő vagy több elem van ugyanabból a fajtából, akkor annyi objektum jön létre, és mindegyik tartalmazza ugyanazokat az adatokat.

Az adatismétlés következményei

- Jelentős redundanciát eredményez.
- Az ár ismétlése várhatóan karbantartási problémához vezet.
- Egy elemre vonatkozó információt tartósan kell tárolni, most objektumhoz kötött.

Az adatismétlés elkerülése

- Az azonos típusú elemeket leíró közös információt egy elkülönített objektumban tároljuk.
- Ezek a „leíró” objektumok nem képviselnek egyedi elemeket, hanem egy csatolt információt tartalmaznak, amely megadja egy elem típusát.
- Nevezzük ezeket az objektumokat *típusnak*.
- Minden egyes elemtípushoz tartozik egy egyedi típus objektum, amely tárolja a nevet, a színt és az árat.
- Az elemeket reprezentáló objektumokban az adatok nem jelennek meg. Ezeket az adatokat a megfelelő típustól kérhetjük le, ezért minden elemnek ismernie kell a megfelelő típust, hivatkoznia kell arra.


: Elem
: Elem
: Elem
: Elem
: Elem
: Típus
név = "2x2 kocka"
szín = Color.RED
ár = 2
: Típus
név = "4x2 kocka"
szín = Color.BLUE
ár = 4

A javítás elemzése

Az előző problémák mindegyikét megoldjuk így, mert:

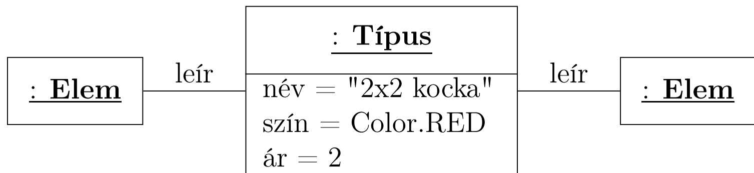
- Az adatokat csak egy helyen tartjuk nyilván, így megszűnik a redundancia.
- Egy adott elem adatainak változtatása egyszerű, csak egy típus adatait kell módosítani.
- A típus mindig létezhet, függetlenül attól, hogy mennyi elem objektum található a rendszerben. Így az információ tárolható még az objektum létrejötte előtt.

Kapcsolatok

- Az új tervben két különböző osztályhoz tartozó objektumok szerepelnek.
  - A típus objektumok azt a statikus információt tartalmazzák, amely minden adott típusú elemre vonatkozik,
  - az elem objektumok egy-egy létező elemet képviselnek.
- A kétféle objektum közötti kapcsolat, hogy egy elem objektum csak a megfelelő típus objektummal együtt használható.
- A kapcsolat a két objektum között az, hogy a típus leírja az elemet.

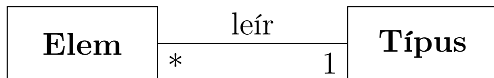
Az objektumok közötti kapcsolatokat az UML-ben összekapcsolásnak nevezzük, amit a kapcsolatban álló objektumokat összekötő vonallal ábrázolunk.

Két 2x2-es kocka és a kapcsolódó típus objektum:



- A kapcsolatok címkézhetőek egy kifejezéssel, amely kifejezi a modellezendő kapcsolatot. (`leír`)
- A címkék rendszerint igék, amelyeket úgy választunk, hogy az összekapcsolt objektumok osztályainak neveivel összeolvasva természetes nyelven kifejezzék a kapcsolat értelmét.
- Ezek a címkék tetszőlegesek, és gyakran el is hagyják ezeket.

- Az objektumok közötti kapcsolatot az osztályok között is feltüntethetjük. (Osztálydiagram, asszociáció.)
- Ez kifejezi, hogy egy adott osztály objektuma és egy másik osztály objektuma kapcsolatban áll egymással.
- Jelölése megegyezik az objektumok esetében megismerttel, csak további jellemzőket tüntethetünk fel a két osztály közötti él mellett.
- Ezek közül az egyik legfontosabb a multiplicitás, azaz, hogy egy adott osztályból mennyi objektum vesz, illetve vehet részt a kapcsolatban.



Kapcsolatok implementációja

- A legtöbb programozási nyelv nem definiálja a kapcsolatok implementációjának módját.
- A legegyszerűbb megközelítés a kapcsolatok kifejezésére általában az, hogy egy összekapcsolt objektumon (osztályon) belül biztosítani kell annak a lehetőségét, hogy az objektum tudja milyen más objektummal áll kapcsolatban.
- A kapcsolat tulajdonságaitól és a használt nyelvtől függően különféleképpen érhetjük ezt el.
- Az egyik legegyszerűbb módszer, ha az objektumok hivatkozásokat (mutató, pointer) tartalmaznak azokra az objektumokra, amelyekkel kapcsolatban állnak.
- Példánkban ez megvalósítható, ha egy elem hivatkozik egy típusra.

Java megvalósítás

```
import java.awt.Color;

public class Típus
{
    private String  név;
    private Color   szín;
    private int     ár;

    public Típus(String név, Color szín, int ár)
    {
        this.név = név; this.szín = szín;   this.ár = ár;
    }

    public String név()     { return név; }
    public Color szín()     { return szín; }
    public int ár()         { return ár; }
}
```

Java megvalósítás (folyt.)

```
import java.awt.Color;

public class Elem
{
    private Típus   típus;

    public Elem(Típus típus)
    {
        this.típus = típus;
    }

    public String név()         { return típus.név(); }

    public Color szín()         { return típus.szín(); }

    public int ár()             { return típus.ár(); }
}
```

Objektumok létrehozása

- Egy elem létrehozásakor biztosítani kell egy hivatkozást a megfelelő típusra.
- Ez nem okozhat gondot, hiszen úgyis csak meghatározott típusú elemeket szeretnénk létrehozni.
- Az elemosztály konstruktora biztosítja, hogy az objektum létrejöttekor a kapcsolat is kialakuljon a megfelelő típus objektummal.

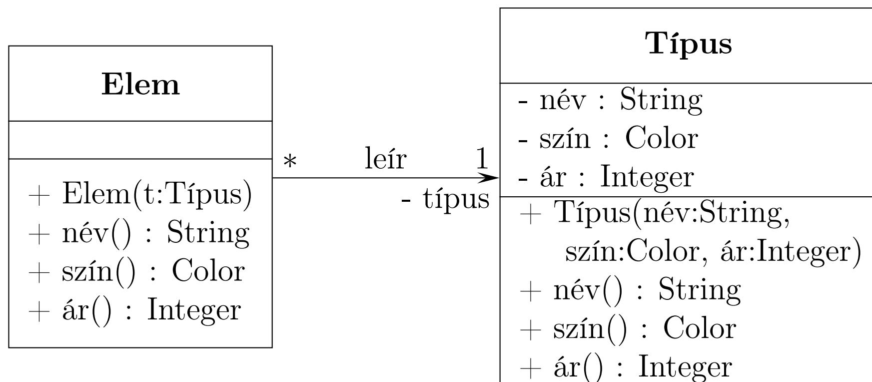
```
Típus piroskocka2x2 = new Típus("2x2 kocka", Color.RED, 2);
Elem e = new Elem(piroskocka2x2);
```

Navigálhatóság

- Az elemek osztálya tartalmaz egy mezőt, amely a megfelelő típus objektumra hivatkozik, de fordítva ez nincs meg: a típusok osztályában nincs utalás az elemekre.
- Egy elem számára elérhető a saját típusa a mutató segítségével, azonban arra nincs lehetőség, hogy közvetlenül megkapjuk egy típushoz kapcsolt elemek halmazát. (Erre a fordított irányú hivatkozásra a használat során nincs is szükség.)
- Ez a kapcsolatot aszimmetrikussá teszi, ami nem szerepelt az előző osztálydiagramban.
- Sokszor nincs szükség a hivatkozásokra mindkét irányban, az egyirányú hivatkozással ugyanakkor jelentős egyszerűsítéseket lehet elérni.

Navigálhatóság (folyt.)

- Azt a tényt, hogy a hivatkozásokat csak az egyik irányban lehet követni (*üzenetküldés*) úgy fejezzük ki, hogy a kapcsolat csak egy irányban navigálható.
- A navigálhatóság egy diagramban úgy jelölhető, hogy egy nyílhegyet teszünk a kapcsolat egyik végére, amely mutatja a navigáció irányát.
- Ha nincs nyílhegy, akkor feltételezzük, hogy a kapcsolat tetszőleges irányban navigálható.



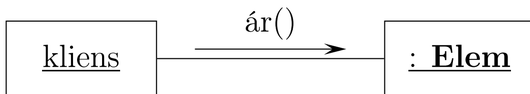
`típus`: szerepnév, ami az implementációban a megfelelő mutatóként jelenik meg. (Láthatóság is megadható.)

Üzenetek

- Az objektumok üzenetek segítségével kommunikálnak, igényelnek másik objektumtól adatot, vagy tevékenység végrehajtást.
- Egy üzenetet egy objektum küld egy másik objektumnak.
- Speciális objektum lehet a kliens, ami egy „külső” elemet jelöl, és az első üzenet küldője.
- Az üzenetek megvalósítása egyszerűbb esetekben egy műveletહívást jelent.
- Csak kapcsolatban álló objektumok küldhetnek egymásnak üzenetet, és a küldés irányának meg kell felelni a kapcsolat navigálhatóságának.

Ár meghatározása

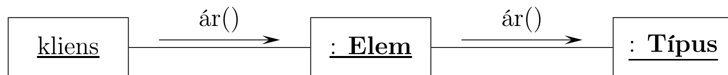
- Egy lehetséges igény egy elem árának lekérdezése.
- A kliens meghívja az elem ár műveletét.
- A kapcsolat melletti címkézett nyíllal jelölhetjük az üzenetküldést.



- Az elvárt reakció, hogy az elem visszaküldi az árát.
- Azonban az ár nem attribútuma az elemnek, így ez esetünkben nem egy saját adat visszaküldését jelenti.

Ár meghatározása (folyt.)

- Az elem nem tárolja az árát, viszont ismeri, hogy melyik objektumtól kérdezheti le.
- Ez a típus objektum, ami kapcsolatban áll az elemmel, és a navigálhatóság az elem felől mutat a típus felé, így a megfelelő művelet meghívható.



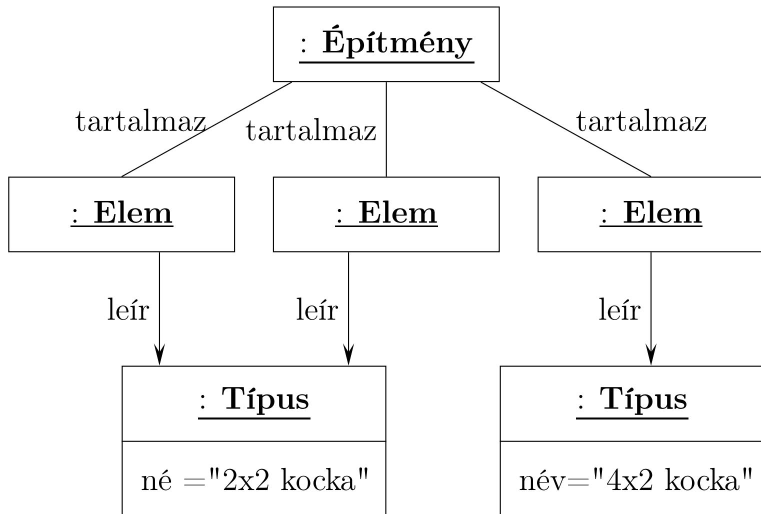
Ez a példa megvilágítja az objektumelvű rendszereknek azt a jellemző vonását, miszerint az adatok szét vannak osztva az összekapcsolt objektumok hálózatában. Az adatok egy része attribútumként áll rendelkezésre, míg más adatokat más objektumoktól kell lekérdezni, amelyekkel az objektum kapcsolatban áll.

Az is látható, hogy egy üzenetküldés több üzenet küldését eredményezheti, ha annak kielégítéséhez több objektum együttműködése szükséges.

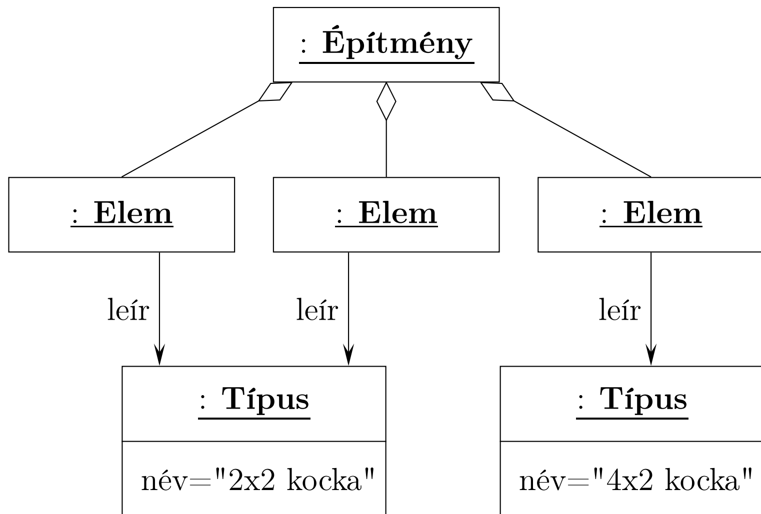
Építmények

- A programnak építményekkel is dolgoznia kell.
- Az elem adatok nyilvántartása mellett, kezelni kell azt is, hogy az elemekből miként épülnek fel az építmények.

Egy egyszerű építmény

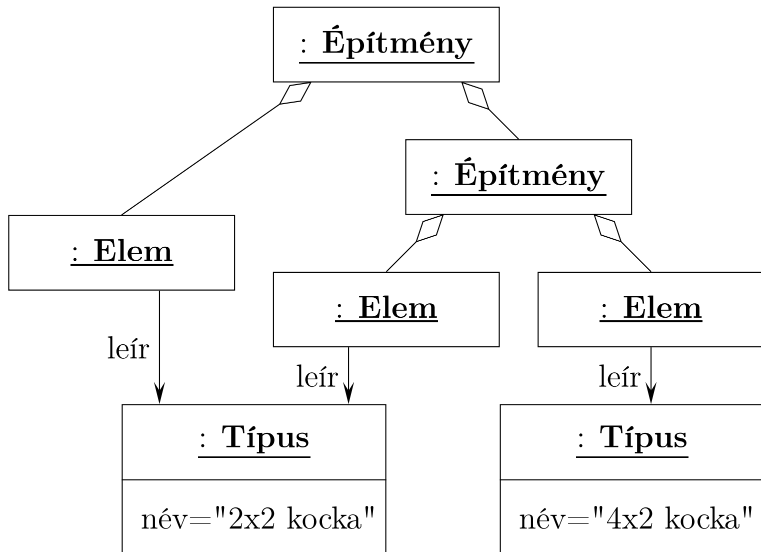


Aggregáció

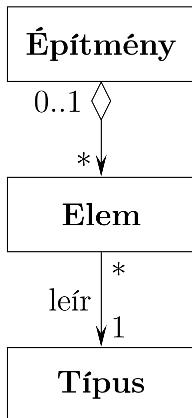


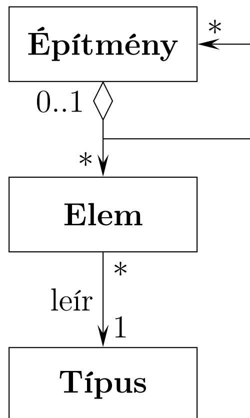
Részépítmények

- Nem elég, hogy egy építményt egyszerűen elemek gyűjteményeként modellezzünk.
- Egy építmény szerkezete hierarchikus is lehet, azaz szerepelhetnek benne részépítmények.
- Az előző építmény másképp is felépíthető (nem feltétlen életszerű).


: Építmény
: Építmény
: Elem
: Elem
: Elem
leír
leír
leír
: Típus
név="2x2 kocka"
: Típus
név="4x2 kocka"

- Ez az aggregációs kapcsolat is csak egy irányban navigálható.
- Az aggregációs kapcsolatok nem csak egy építményt kapcsolhatnak össze elemekkel, hanem építmények között is kapcsolatot teremthetnek.
- Az UML-ben a kapcsolatok típusosak, azaz egyező címkéjű kapcsolatoknak ugyanolyan típusú objektumokat kell összekötniük.
- Az utóbbi esetben az aggregációs kapcsolat egyik felén nem egyetlen osztályhoz tartozó objektumok szerepelhetnek, hanem több osztály objektumai fordulhatnak ott elő.

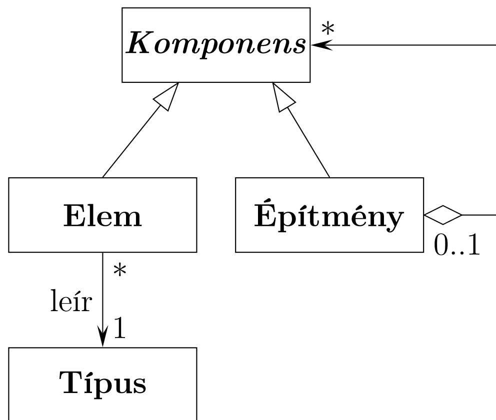

1. eset
Építmény
0..1
*
Elem
*
leír
1
Típus


2. eset
Építmény
*
0..1
*
Elem
*
leír
1
Típus

Polimorfizmus

- Az UML típusosságát azonban fenn kell tartani, ezért meg kell határoznunk azon osztályok halmazát, amelyek részt vehetnek a kapcsolatban.
- Létre kell hoznunk egy általános osztályt, amelynek speciális esetei lehetnek a meghatározott halmazba tartozó osztályok.
- Ezután ezt az általános osztályt használhatjuk a kapcsolat megfelelő oldalán.
- Vezessük be a `Komponens` osztályt, amelynek specializációja lehet az `Elem` vagy az `Építmény` osztály.

Specializáció



Öröklődés

- A specializáció megvalósítására az objektumelvű nyelvekben az öröklődés szolgál.
- Definiálhatunk egy osztályt, majd ebből az osztályból öröklődéssel származtathatunk újabb osztályokat.
- A származtatott osztályok példányai bárhol előfordulhatnak, ahol az eredeti osztály példányai szerepeltek.

Komponens osztály

- Absztrakt osztály, nincs ilyen objektum.
- A származtatott osztályok közös tulajdonságait határozza meg. (Most csak műveletek.)
- Absztrakt műveletek:
  - ár lekérdezése (`ár`),
  - komponens beillesztése (`betesz`).
- Az absztrakt műveleteket definiálják a származtatott osztályok.